MIIC REST API

Version 1.0

The MIIC REST API was developed in parallel with the MIIC web-application and is designed to offer (essentially) the same feature set to software clients as normal web users.

This version of the REST API supports all tasks associated with creating and executing inter-calibration plans (ICPlans). Users can add, retrieve, modify and delete ICPlans, start/stop ICPlan execution, and retrieve collected ICPlan data as NetCDF files. ICPlans (and other associated objects) are represented in XML format.

The REST API also includes a feature to retrieve satellite ephemeris data (position & velocity) for any satellite on any day. This may be useful to clients who wish to visualize satellite orbits. Note that users who want satellite TLEs should use www.spacetrack.org.

Note that MIIC does not currently require clients to obtain any "secret" API key in order to use the REST interface. However, clients must still authenticate via the NASA URS system. That means you must have a NASA URS account, and you must first authorize the MIIC application by logging in to the MIIC webpage.

For your RESTful convenience, MIIC provides two separate software clients:

1. A Java client. This includes "JAXB" transformations of XML to and from Java objects, so no XML parsing is necessary!
2. A "bash" shell client that is based on cURL. You could probably use the cURL library available in other programming languages like python too.

The "Bash" Client

Bash scripts can be used to make REST calls to MIIC server. The scripts require an up-to-date version of the cURL library (7.30.0 or newer) on your system. Older versions can't retrieve credentials from a .netrc file or perform the necessary redirects during authentication.

There is a setup script (*miicrestsetup.sh*) you must run one time in order to prepare your environment. Other scripts make HTTP calls to MIIC URLs. The script must match the HTTP request type from the REST API.

The scripts are:

- miicrestget.sh for GET requests
- miicrestpost.sh for POST requests
- miicrestput.sh for PUT requests
- miicrestdelete.sh for DELETE requests

The scripts use secure redirects to NASA URS to perform authentication. Session tokens are stored in the file ~/.urs_cookies. All scripts return "0" if successful. If an HTTP error is encountered, a message is printed to standard error, and the HTTP error code is returned (404, etc).

All scripts require a MIIC URL. You can supply the whole URL on the command line, like "https://dev-miic.larc.nasa.gov/miic/rest/icplan/17". Or, you can just use the portion of the URL starting with rest, as in "rest/icplan/17". In the case of the shortened URL, the environment variable "MIIC_ROOT" points to the server.

> **Cross-site Request Forgery Protection**
> You may notice that the miicrestpost/put/delete scripts first make a GET call to "miic/ursLogin".
>
> This does not mean that you need to "login" before every POST call. The GET request returns a unique "CSRF" key to the client. This key must be included in POST calls you send to the server (internally, PUT and DELETE also use POST). This is because POST calls actually ***change*** the server and we need to protect against hackers tricking users into clicking malicious links.

REST Setup

The script *miicrestsetup.sh* will prepare your environment for making REST calls to MIIC. Effectively, this means storing your NASA URS credentials in a file in your home directory. The file is plaintext but is only readable by you, so it should be safe. The file is required so that your URS credentials can be sent securely to NASA URS when necessary.

By default miicrestsetup.sh will attempt to authenticate with the production MIIC server at URL https://miic.larc.nasa.gov/miic. To authenticate with another MIIC, set the environment variable MIIC_ROOT to something else.

```
% miicrestsetup.sh

Logging in to MIIC server: https://miic.larc.nasa.gov/miic

Type your URS username, followed by [ENTER]

*****

Type your URS password, followed by [ENTER]

***********

Your URS credentials have been stored to file ~/.netrc
```

GET Requests

The script *miicrestget.sh* sends authenticated HTTP GET requests to MIIC. GET requests return information but don't change anything on the server.

Output is saved to a provided filename or written to standard out.

For example, to get all the plans owned by user chris, output to file allplans.xml

```
% miicrestget.sh http://miic.larc.nasa.gov/miic/rest/icplans/chris allplan
s.xml
```

POST Requests

POST requests create new objects on the server. The server response is written to the optional file or to standard out.

For example, send a plan defined in file myPlan.xml to the server, and save the return value (the new ID) to file planID.out:

```
% miicrestpost.sh http://miic.larc.nasa.gov/miic/rest/icplan myPlan.xml
planID.out
```

PUT Requests

PUT requests change existing objects on the server. The server response is written to the optional file or standard out. The script sends a POST request with a special tag to identify it as a logical PUT request at the server.

For example, to send an updated plan defined in planUpdate.xml to the server, and print the result to standard out:

```
% miicrestput.sh http://miic.larc.nasa.gov/miic/rest/icplan planUpdate.xml
```

DELETE Requests

DELETE requests remove objects from the server. The script sends a POST request with a special tag to identify it as a logical DELETE request at the server.

For example, to delete plan ID 17:

```
% miicrestdelete.sh http://miic.larc.nasa.gov/miic/rest/icplan/17
```

JAVA Client

A Java-based REST client in the archive "rest-client.jar" can simplify writing Java clients that talk to MIIC.

The best way to use this client is in conjunction with the spring framework RestTemplate class. Other libraries will probably work with the Java Client, but RestTemplate manages URS authentication & tokens, and the conversion of raw XML to and from Java objects.

This section includes some code to get you started writing a Java client using spring RestTemplate.

Prepare RestTemplate Object

First, create a RestTemplate object that knows how to authenticate with NASA URS by using your URS credentials stored at the file ~/.netrc. You already have this file if you ran the *miicrestsetup.sh* script defined above.

```
Login netlogin = Netrc.getLogin("urs.earthdata.nasa.gov");
AuthHttpComponentsClientHttpRequestFactory requestFactory = new
AuthHttpComponentsClientHttpRequestFactory(
        new HttpHost("urs.earthdata.nasa.gov",443,"https"), netlogin.user,
netlogin.password);
RestTemplate rest = new RestTemplate(requestFactory);
```

Next, add message converters to convert between raw HTML payloads and more convenient things like Java objects:

```
List<HttpMessageConverter<?>> converters = new
ArrayList<HttpMessageConverter<?>>();
converters.add(new StringHttpMessageConverter());
converters.add(new Jaxb2RootElementHttpMessageConverter());
converters.add(new MappingJackson2HttpMessageConverter());
converters.add(new ResourceHttpMessageConverter());
rest.setMessageConverters(converters);
```

Here is what the message converters do in a nutshell:

- StringHttpMessageConverter – text payload Java String object
- Jaxb2RootElementHttpMessageConverter – XML payload Java objects
- MappingJackson2HttpMessageConverter – JSON payload Java objects
- ResourceHttpMessageConverter - binary payload Java byte stream

Make a GET Request

The Java client includes generated JAX-B objects that simplify sending and receiving objects to MIIC.

For example, retrieve ICPlan 11 and print out the duration of each event in seconds:

```
IcPlan plan =
rest.getForObject("https://miic.larc.nasa.gov/rest/icplan/11",
IcPlan.class);
for (IcEvent event : plan.getEvents().getEvent()) {

System.out.println((event.getEnd().toGregorianCalendar().getTimeInMillis()-


event.getBegin().toGregorianCalendar().getTimeInMillis())/1000);
}
```

RESTful Inter-calibration Plans

The REST interface is primarily used to define and execute ICPlans.

ICPlans execute at the server by performing these four tasks in-order:

1. ***PREDICT*** - find inter-calibration events (ICEvents) of interest, defined by space/time bounds.
2. ***LOCATE*** - find all files that contain data for your ICEvents
3. ***COLLECT*** - collect data for ICEvents
4. ***ANALYZE*** - analyze collected data

ICPlans always have a date range, a *target* entity and may also have a *reference* entity. The date range is a single continuous time interval over which to find events and collect data.

Target and reference entities are used during event prediction and data collection. At present they may be one of:

- Satellite: a spacecraft (e.g. AQUA)
- Instrument: the instrument abord a spacecraft (e.g. MODIS on AQUA)
- Data collection: a data product derived from instrument data (e.g. MODIS_Aqua_MYD02SS1_C6)

Event Prediction

ICPlans with a target entity support simple data mining tasks:

- The LEO/ground predictor is able to predict ICEvents of a LEO spacecraft vs. a ground station (defined by a point and radius).
- The ASDC predictor is able to use enhanced metadata to predict ICEvents for CALIPSO and CERES data products vs. a rectangular lat/lon box.

ICPlans with both target and reference entities support data mining between two spacecraft:

- The LEO/GEO predictor is able to predict events when a LEO spacecraft crosses into a GEO satellite cone.
- The LEO/LEO predictor is able to predict events when a LEO spacecraft crosses into a tent structure of another LEO spacecraft

Data Location & Collection

In order to locate data files and collect data, ICPlans must supply a data collection entity for target and/or reference. ICPlans using satellite or instrument entities can't collect data.

ICPlans with data collection entities as target and/or reference are able to collect data:

- The 2DHistogram data collector is able to average parameters from data collections over the events' geographic and/or time bounds. The histogram axes are spatial by default but do not need to be spatial.
- The Tuple data collector is able to filter parameters from data collections over the events' geographic and/or time bounds.

ICPlan State

ICPlan state is one of the following: INITIAL, PREDICTING_EVENTS, PREDICTION_FAILED, EVENTS_PREDICTED, LOCATING_DATA, LOCATION_FAILED, DATA_LOCATED, ACQUIRING_DATA, ACQUISITION_FAILED, DATA_ACQUIRED, ANALYZING, ANALYSIS_FAILED, ANALYSIS_COMPLETED

The green and red states are **terminal**. A plan that is not currently executing must be in one of those states.

The red states are **failure** states. To enter one of these states, the system must have encountered an error making it unable to continue processing, or the user stopped an in-progress execution.

- PREDICTION_FAILED: no inter-comparison events were found or there was an error running the event prediction (e.g. unable to retrieve any satellite TLE files)
- LOCATION_FAILED: **none** of the required data files could be found. This is usually caused by OPeNDAP servers not having the files you need. (if at least some files were found, we keep going...)
- ACQUISITION_FAILED: there was an error collecting data. It will attempt to collect everything it can, but enter this state if anything failed.
- ANALYSIS_FAILED: there was an error performing analysis.

To enter a green **completion** state, the system must be "done" processing your plan. This happens when:

- INITIAL: the plan was created or modified but not yet executed
- EVENTS_PREDICTED: the user didn't provide data variables, or asked to stop executing after the "PREDICT" task
- DATA_LOCATED: the user asked to stop executing after the "LOCATE" task
- DATA_ACQUIRED: the didn't ask for any analyses, or asked to stop executing after the "ACQUIRE" task.

> **missing data!**
> Please note that there may be potentially many location failures, and your plan will keep executing! This is because we can't always have all data collection files available at OPeNDAP servers but we still want to collect all that we can.

ICEvent State

ICEvent state will be one of the following: INITIAL, LOCATING_DATA, LOCATION_FAILED, DATA_LOCATED, ACQUIRING_DATA, ACQUISITION_FAILED, DATA_ACQUIRED

The green and red states are **terminal**. Events in a plan that is not executing must be in one of those states.

To enter a red **failure** state, the system must have encountered an error making it unable to continue processing the event. This happens when:

- LOCATION_FAILED: A required target and/or reference data file is missing.
- ACQUISITION_FAILED: There was an error collecting data (network down or server error)

To enter a green **terminal** state, the system must be "done" processing the event. This happens when:

- INITIAL: An *event predictor* generated the event, but we have not yet attempted the LOCATE or COLLECT tasks.
- DATA_LOCATED: the user asked to stop executing after the "LOCATE" task
- DATA_ACQUIRED: the data has been retrieved

XML Object Definitions

XML format is used to define objects sent to and from the server. The primary objects of importance are:

- Inter-calibration plans (ICPlans)
- Inter-calibration events (ICEvents)
- ICPlan Analysis
- Data collections

ICPlan XML

Here are the major elements of an intercalibration plan XML document along with their description, shown as an example document.

Please note:

- ICPlan elements noted as "server generated" are filled by the server when it executes your plan. These can never be included in ICPlans that you send to the server, they are only part of ICPlans you retrieve from the server.
- ICPlan elements noted as "client required" must always be defined in ICPlans that you send to the server.
- ICPlan elements noted as "client optional" may be defined in ICPlans that you send to the server.

icPlan Element

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<icPlan>

    <!-- the date range of the ICPlan (client required) -->
    <beginDate>2010-12-31T19:00:00-05:00</beginDate>
    <endDate>2011-01-01T19:00:00-05:00</endDate>

    <!-- non-unique name (client required) -->
    <name>RESTful plan test</name>

    <!-- target selection (client optional) -->
    <!-- clients may provide one of: satellite, instrument, collection -->
    <targetSatellite>AQUA</targetSatellite>
    <targetInstrument>CERES-FM3</targetInstrument>
    <targetCollection>CER_SSF_Aqua-FM3-MODIS</targetCollection>

    <!-- auto-execute the plan when its definition changes, default false
(client optional) -->
    <active>false</active>

    <!-- reference selection (client optional) -->
    <!-- clients may provide one of: satellite, instrument, collection -->
    <referenceSatellite>AQUA</referenceSatellite>
    <referenceInstrument>CERES-FM3</referenceInstrument>
    <referenceCollection>CER_SSF_Aqua-FM3-MODIS</referenceCollection>


    <reference>modis_l1b_aqua</reference>

    <!-- variables to collect from target (client optional) -->
    <targetVariables>
```

```
       <dataVariableRef>

          <!-- Optional constraint expression to limit the data retrieved
from a variable
               The format of a dimension constraint is
"[start:stride:stop]", where * may be used as a stop value.
               If you use a constraint expression, you must constrain ALL
dimensions of the variable, in order.
               So if a variable had two dimensions and you wish to stride
the first by 10, use: "[0:10:*] [0:1:*]" -->
          <constraintExpression>[0:10:*]</constraintExpression>

          <!-- This section identifies the variable.
               IDs are obtained from the Data Collection XML document. -->
          <dataVariableId>

<name>Full_Footprint_Area_Column_averaged_relative_humidity</name>
              <collectionName>CER_SSF_Aqua-FM3-MODIS</collectionName>
          </dataVariableId>
       </dataVariableRef>
    </targetVariables>

    <!-- variables to collect from reference (client optional) -->
    <referenceVariables>...</referenceVariables>

    <!-- plan options: the event predictor and data collector (client
optional) -->
    <options>
        <entry>
            <key>eventPredictorID</key>
            <value>LeovsGeo</value>
        </entry>
        <entry>
            <key>dataCollectorID</key>
            <value>2DHistogramAverages</value>
        </entry>
    </options>

    <!-- XML comments describing options for the current event predictor
(server generated) -->
    <eventPredictionOptionsHelp>...</eventPredictionOptionsHelp>

    <!-- user options for event prediction (client optional) -->
    <eventPredictionOptions>...</eventPredictionOptions>

    <!-- XML comments describing options for the current data collector
(server generated) -->
    <dataCollectionOptionsHelp>...</dataCollectionOptionsHelp>

    <!-- user options for data collection (client optional) -->
    <dataCollectionOptions>...</dataCollectionOptions>

    <!-- plan state (server generated) -->
```

```
<!-- INITIAL, PREDICTING_EVENTS, PREDICTION_FAILED, EVENTS_PREDICTED,
     LOCATING_DATA, LOCATION_FAILED, DATA_LOCATED,
     ACQUIRING_DATA, ACQUISITION_FAILED, DATA_ACQUIRED,
     ANALYZING, ANALYSIS_FAILED, ANALYSIS_COMPLETED -->
<state>LOCATION_FAILED</state>

<!-- unique ID (server generated) -->
<id>5</id>

<!-- owner name (server generated) -->
<owner>aron</owner>

<!-- date created (server generated) -->
<created>2015-04-27T13:49:05.632-04:00</created>

<!-- last error (server generated) -->
<error>No servers have files for the collection...</error>

<!-- inter-calibration results, see below (server generated) -->
<events>...</events>
```

```
</icPlan>
```

A note on dates
ICPlan begin/end dates are sent along with timezone information (e.g. the -05:00 in the accompanying XML). At the server, dates are converted to GMT.

In this example, "2010-12-31T19:00:00-05:00" represents a clock 5 hours behind GMT, resulting in a GMT server time of 2011-01-01T00:00:00.

At the server, ICPlan dates only use calendar day, so dates sent to the server are always converted to "midnight GMT" on that day.

A REST client currently has no way of querying the system for what *event predictors* or *data collectors* are available to use.

In the meantime, use these documents: Event Predictors Data Collectors

ICEvent XML

Here are the elements of an intercalibration event element along with their description. Note that the types of information therein depends on the ICPlan state:

- STATE == INITIAL: no ICEvents
- STATE >= EVENTS_PREDICTED: geo and time extents
- STATE >= DATA_LOCATED: source URLs
- STATE >= DATA_COLLECTED: query & data URLs

event Element

```
 <event>
  <!-- INITIAL, LOCATING_DATA, LOCATION_FAILED, DATA_LOCATED,
ACQUIRING_DATA, ACQUISITION_FAILED, DATA_ACQUIRED -->
  <state>DATA_ACQUIRED</state>

  <!-- geo extents of the event -->
  <latNorth>-32.95445463913686</latNorth>
  <latSouth>-84.79223820801909</latSouth>
  <lonEast>74.62404595415231</lonEast>
  <lonWest>-33.182695945018395</lonWest>

  <!-- time extents of the event -->
  <begin>2013-01-05T09:39:50-06:00</begin>
  <end>2013-01-05T09:56:41-06:00</end>

  <!-- unique event ID -->
  <id>606</id>

  <!-- data collected by the server -->
  <targetData>

    <!-- unique data ID -->
    <id>1796</id>

    <!-- the format of NetCDF data -->
    <type>HISTOGRAM_2D_AVG</type>

    <!-- server URL to retrieve this data as NetCDF -->
    <dataURL>http://miic1.larc.nasa.gov:8080/miic/rest/...</dataURL>

    <!-- OPeNDAP files used in this event -->
    <sourceURLs>
      <URL>http://miic1.larc.nasa.gov:8080/opendap/hyrax/CERES...</URL>
    </sourceURLs>

    <!-- OPeNDAP server-side function used to collect data -->
    <queryURL>?histogram_2d_avg(x_axis(...</queryURL>
  </targetData>

  <!-- reference data collected by the server -->
  <referenceData>
    <id>1797</id>
    <type>HISTOGRAM_2D_AVG</type>
    <dataURL>http://miic1.larc.nasa.gov:8080/miic/rest/...</dataURL>
    <sourceURLs>
      <URL>http://miic1.larc.nasa.gov:8080/opendap/hyrax/VIIRS...</URL>
      ...
    </sourceURLs>
    <queryURL>?histogram_2d_avg(x_axis(Longitude_Sub,...</queryURL>
  </referenceData>

</event>
```

Analysis XML

Analysis XML documents indicate which analysis plugin to run and the arguments to pass to the plugin.

analysis Element

```
<analysis>
   <!-- state, set by server -->
   <state>DATA_ANALYZED</state>

   <!-- unique analysis ID, set by server -->
   <id>54</id>

   <!-- if an error when running plugin -->
   <error>...</error>

   <!-- path to analysis objects generated by plugin, in AIDA results tree
-->
   <path>/1D Histogram/Latitude_Sub</path>

   <!-- name and version of analysis plugin to use -->
   <pluginName>Statistics</pluginName>
   <pluginVersion>1.0</pluginVersion>

   <!-- arguments to pass to the analysis plugin -->
   <pluginArgs>-x_var TARGET:Latitude_Sub </pluginArgs>

   <!-- corresponding ICPlan ID -->
   <planId>11</planId>
</analysis>
```

Analysis plugins read information from the ICPlan and generate AIDA analysis objects (histograms, point sets, etc.)

At the present time, analysis results are not included in the XML document.

To retrieve analysis results, clients have these options:

1. Retrieve the AIDA results tree from the server. The client must have other software that is able to read AIDA objects.
2. Retrieve JSON-formatted objects from the AIDA results tree. We currently support 1D & 2D histograms and profiles.

Data Collection XML

Data Collection documents describe the collections supported by the system and their data variables.

Data variables provide metadata which may be useful to some clients including dimension names, legal ranges, and missing values.

dataCollections Element

```
<dataCollections>
   <satelliteDataCollection>
      <!-- name of data collection -->
      <name>CER_SSF_Aqua-FM3-MODIS</name>

      <!-- filename patterns -->
      <fileDuration>3600</fileDuration>

<fileNamePattern>CER_SSF_Aqua-FM3-MODIS_Edition3A_\d{6}\.(\d{4}\d{2}\d{4})
</fileNamePattern>

<fileStartDatePattern>CER_SSF_Aqua-FM3-MODIS_Edition3A_\d{6}\.(\d{4}\d{2}\
d{4})</fileStartDatePattern>
      <fileStartDateFormat>yyyyMMddHH</fileStartDateFormat>

      <!-- available data variables -->
      <dataVariables>
         <dataVariable>

            <!-- IDs are used to specify variables in an ICPlan XML
document -->
            <dataVariableId>
               <name>Time_and_Position_Time_of_observation</name>
               <collectionName>CER_SSF_Aqua-FM3-MODIS</collectionName>
            </dataVariableId>

            <!-- Known metadata for this variable -->
            <units>day</units>
            <dims>
               <name>Footprints</name>
               <size>91172</size>
            </dims>
            <validMin>2440000.0</validMin>
            <validMax>2480000.0</validMax>
            <missingValue>1.7976931348623157E308</missingValue>
         </dataVariable>
         ...
      </dataVariables>

      <!-- the instrument & spacecraft upon which this collection is
derived -->
      <instrument>
         <name>CERES-FM3</name>
         <satellite>
            <launch/>
            <name>AQUA</name>
            <satID>27424</satID>
            <type>LEO</type>
         </satellite>
         <swathAngle>110.0</swathAngle>
     </instrument>
   </satelliteDataCollection>
</dataCollections>
```

REST API

Methods Returning XML Objects

GET rest/icplans/{uid}

Retrieves all ICPlans owned by one user.

Expected code: 200 (OK)

Expected response: ICPlan XML

GET rest/icplans/find?uid={user name}&name{plan name}

Find ICPlans by user name and/or plan name. Plan names may use % for wildcard matching.

Expected code: 200 (OK)

Expected response: a list of integer ICPlan IDs matching the query

GET rest/icplan/{planID}

Retrieve one ICPlan, by ID.

Expected code: 200 (OK)

Expected response: ICPlan XML

GET rest/planExecutionState/{planID}

Returns true if the plan is still executing

Expected code: 200 (OK)

Expected response: "true" or "false"

GET rest/dataCollections

Returns list of all data collections.

Expected code: 200 (OK)

Expected response: Data Collection XML

GET rest/icPlanAnalyses/{planID}

Returns analyses for the requested ICPlan.

Expected code: 200 (OK)

Expected response: Analysis XML

Methods Returning JSON Objects

GET ajax/IHistogram1D/{planID}/?path={ITree Path}

Retrieve a 1D Histogram object. ITree path may be obtained from the Analysis XML document.

Expected code: 200 (OK)

Expected response: JSON data

Sample JSON for 1D Histogram


```
{
  "count" : 2238865,
  "mean" : 87.11295092953549,
  "std_dev" : 12.684365658136658,
  "counts" : [ 145, 210202, 2028518 ],
  "errors" : [ 12.041594578792296, 458.477916589229,
1424.260509878723 ],
  "means" : [ 32.53698041192416, 54.549699907855185,
90.49116793643923 ],
  "annotation" : {
    "Title" : "Unfiltered_Radiances_CERES_LW_radiance___upwards",
    "AidaPath" : "/1D
Histogram/Unfiltered_Radiances_CERES_LW_radiance___upwards",
    "FullPath" : "/Analysis Results for NINO3-Jan-July_2015/1D
Histogram/Unfiltered_Radiances_CERES_LW_radiance___upwards",
    "xAxisLabel" :
"Unfiltered_Radiances_CERES_LW_radiance___upwards",
    "xUnits" : "Watts per square meter per steradian"
  },
  "xAxis" : {
    "binWidth" : 33.333333333333336,
    "centers" : [ 16.666666666666668, 50.0, 83.33333333333334 ]
  }
}
```

GET ajax/IProfile1D/{planID}/?path={ITree Path}

Retrieve a 1D Profile object. ITree path may be obtained from the Analysis XML document.

Expected code: 200 (OK)

Expected response: JSON data

Sample JSON for 1D Profile


```
{
  "count" : 2324278,
  "mean" : 300.59698478572284,
  "std_dev" : 1.1272463884642834,
  "counts" : [ 782688, 766335, 775255 ],
  "errors" : [ 0.8908053673182319, 0.6816137243080547,
0.9140478015897495 ],
  "means" : [ 299.53731260116814, 301.18276478125756,
301.0877768346701 ],
  "annotation" : {
    "Title" : "Full_Footprint_Area_Surface_skin_temperature",
    "AidaPath" : "/1D
Profile/x=Time_and_Position_Time_of_observation/Full_Footprint_Are
a_Surface_skin_temperature",
    "FullPath" : "/Analysis Results for NINO3-Jan-July_2015/1D
Profile/x=Time_and_Position_Time_of_observation/Full_Footprint_Are
a_Surface_skin_temperature",
    "xAxisLabel" : "Time_and_Position_Time_of_observation",
    "xUnits" : "day",
    "yAxisLabel" : "Full_Footprint_Area_Surface_skin_temperature",
    "yUnits" : "Kelvin"
  },
  "xAxis" : {
    "binWidth" : 70.36415992304683,
    "centers" : [ 2457059.5319604822, 2457129.8961204053,
2457200.2602803283 ]
  }
}
```

GET ajax/IHistogram2D/{planID}/?path={ITree Path}

Retrieve a 2D Histogram object. ITree path may be obtained from the Analysis XML document.

Expected code: 200 (OK)

Expected response: JSON data

Sample JSON for 2D Histogram


```
{
  "count" : 2238865,
  "xMean" : 89.93232255599716,
  "yMean" : 87.11295092953549,
  "xStd_dev" : 2.883749101909602,
  "yStd_dev" : 12.684365658221834,
  "counts" : [ [ 56498, 1085123 ], [ 7582, 1089662 ] ],
  "errors" : [ [ 237.6930794112441, 1041.6923730161416 ], [
87.07468059085832, 1043.8687656980642 ] ],
  "annotation" : {
    "Title" :
"Time_and_Position_Colatitude_of_CERES_FOV_at_surface vs
Unfiltered_Radiances_CERES_LW_radiance___upwards",
    "AidaPath" : "/2D
Histogram/Time_and_Position_Colatitude_of_CERES_FOV_at_surface vs
Unfiltered_Radiances_CERES_LW_radiance___upwards",
    "FullPath" : "/Analysis Results for NINO3-Jan-July_2015/2D
Histogram/Time_and_Position_Colatitude_of_CERES_FOV_at_surface vs
Unfiltered_Radiances_CERES_LW_radiance___upwards",
    "xAxisLabel" :
"Time_and_Position_Colatitude_of_CERES_FOV_at_surface",
    "xUnits" : "degrees",
    "yAxisLabel" :
"Unfiltered_Radiances_CERES_LW_radiance___upwards",
    "yUnits" : "Watts per square meter per steradian"
  },
  "xAxis" : {
    "binWidth" : 4.999988555908217,
    "centers" : [ 87.5000171661377, 92.5000057220459 ]
  },
  "yAxis" : {
    "binWidth" : 50.0,
    "centers" : [ 25.0, 75.0 ]
  }
}
```

GET ajax/IProfile2D/{planID}/?path={ITree Path}

Retrieve a 2D Profile object. ITree path may be obtained from the Analysis XML document.

Expected code: 200 (OK)

Expected response: JSON data

Sample JSON for 2D Profile


```
{
  "count" : 2238865,
  "zStd_dev" : 12.684365658221976,
  "zMean" : 87.11295092953549,
  "xMean" : 240.56122886718254,
  "yMean" : 89.93232255599716,
  "xStd_dev" : 17.129898578274894,
  "yStd_dev" : 2.883749101910548,
  "counts" : [ [ 563825, 526229 ], [ 577796, 571015 ] ],
  "errors" : [ [ 1.488725198136091E-4, 1.7307781158015217E-4 ], [
1.4397653717681196E-4, 1.586013967606676E-4 ] ],
  "means" : [ [ 83.93804848390815, 91.07856371001189 ], [
83.18906727461325, 90.56377657129262 ] ],
  "annotation" : {
    "Title" : "Unfiltered_Radiances_CERES_LW_radiance___upwards",
    "AidaPath" : "/2D
Profile/x=Time_and_Position_Longitude_of_CERES_FOV_at_surface
y=Time_and_Position_Colatitude_of_CERES_FOV_at_surface/Unfiltered_
Radiances_CERES_LW_radiance___upwards",
    "FullPath" : "/Analysis Results for NINO3-Jan-July_2015/2D
Profile/x=Time_and_Position_Longitude_of_CERES_FOV_at_surface
y=Time_and_Position_Colatitude_of_CERES_FOV_at_surface/Unfiltered_
Radiances_CERES_LW_radiance___upwards",
    "xAxisLabel" :
"Time_and_Position_Longitude_of_CERES_FOV_at_surface",
    "xUnits" : "degrees",
    "yAxisLabel" :
"Time_and_Position_Colatitude_of_CERES_FOV_at_surface",
    "yUnits" : "degrees"
  },
  "xAxis" : {
    "binWidth" : 30.000000000000043,
    "centers" : [ 225.0, 255.00000000000003 ]
  },
  "yAxis" : {
    "binWidth" : 4.999988555908217,
    "centers" : [ 87.5000171661377, 92.5000057220459 ]
  }
}
```

Methods Returning Other Data

GET rest/icplandata/{planID}

Retrieves a zip file containing all NetCDF data files collected for the plan.

Expected code: 200 (OK)

Expected response: ZIP file

GET rest/icplanresults/{planID}

Retrieves a zip file containing the AIDA analysis results for the plan.

Expected code: 200 (OK)

Expected response: ZIP file

GET rest/icplandata/?dataID={dataID}

Retrieve one NetCDF data file from an ICPlan. DataIDs are obtained from ICPlan XML.

Expected code: 200 (OK)

Expected response: ZIP file

GET /rest/ephemeris/{date}/{satid}/{dt}

Generate satellite ephemeris data.

- date: ephemeris data returned for one calendar day (24 hour period)
- satid: satellite identifier (used to obtain TLE file)
- dt: unit of orbit propagation, in seconds

Expected code: 200 (OK)

Expected response: ZIP file

Methods Adding New XML Objects

POST rest/icplan

Create a new ICPlan on the server.

The system will automatically execute the new plan if its *active* field is set to true.

POST Body: ICPlan XML

Expected code: 200 (OK)

Expected response: the ID of your new plan

POST rest/analysis

Create a new analysis for an ICPlan.

The system will automatically execute the analysis (provided data can be collected and the plan was set to *active*).

POST Body: Analysis XML

Expected code: 200 (OK)

Expected response: the ID of your new analysis.

Methods Modifying Existing XML Objects

PUT rest/icplan

Modify an existing ICPlan on the server.

In the XML, the planID element must be set, so we know which plan to modify.

> **Ignores some ICPlan fields...**
> Server-generated elements in the plan will be ignored (e.g. events).

PUT Body: ICPlan XML

Expected code: 200 (OK)

PUT rest/analysis

Modify an existing analysis task for an ICPlan.

The analysisID element must be set, so we know which one to modify.

PUT Body: Analysis XML

Expected code: 200 (OK)

PUT rest/startPlanExecution/{planID}?stopAfter={TASK}

Start executing the ICPlan. Optionally, stop after the desired task.

Valid tasks are: PREDICT, LOCATE, COLLECT, ANALYZE

Expected code: 200 (OK)

PUT rest/stopPlanExecution/{planID}

Stop executing the ICPlan.

Expected code: 200 (OK)

Methods Deleting XML Objects

DELETE rest/icplan/{planID}

Removes an intercalibration plan from the server, and deletes any associated file storage.

> **Stop the plan first!**
> You can't delete a plan that is executing, so stop it first.

Expected code: 200 (OK)

DELETE rest/analysis/{analysisID}

Removes an analysis object from the system.

Expected code: 200 (OK)